THESIS

ROBUSTNESS: A BETTER MEASURE OF ALGORITHM PERFORMANCE

by

Roger D. Musselman

September 2007

Thesis Advisor:	Paul J. Sanchez
Second Reader:	Susan M. Sanchez

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Robustness: A Better Measure of Algorithm Performance			**5. FUNDING NUMBERS**
6. AUTHOR(S) Roger D. Musselman			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Algorithms are an essential part of Operations Research (OR) methodology. Therefore, the efficiency of the algorithms must be a consideration. However, traditional approaches to assessing algorithm efficiency do not always captured the real-world trade-offs involved. This thesis explored the use of a new measure of algorithm efficiency, robustness, and contrasted it with the traditional "big-O" analysis. Sorting algorithms were used to illustrate the trade-offs.

The use of Dr. Genichi Taguchi's robust design techniques allowed us to take into account the impact of factors which would be uncontrollable in the real world, by measuring how those factors affect the consistency of the results. These factors, which are treated separately by big-O analysis, are incorporated as an integral part of robust analysis. The hypothesis was that robustness is potentially a more useful description of algorithm performance than the more traditional big-O analyses.

The results of experimentation supported this hypothesis. Where big-O analysis only considers the average performance, robustness integrates the average performance and the consistency of performance. Most importantly, the robust analysis we performed yielded results that are consistent with actual usage—practitioners prefer quicksort over heap sort, despite the fact that under big-O analysis heap sort dominates quicksort.

14. SUBJECT TERMS Robust Design, Taguchi Method, Sorting Algorithms, Quicksort, Heap Sort			**15. NUMBER OF PAGES** 75
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

ROBUSTNESS: A BETTER MEASURE OF ALGORITHM PERFORMANCE

Roger D. Musselman
Lieutenant Commander, United States Navy
B.S., Rensselaer Polytechnic Institute, 1994

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September 2007

Author:	Roger D. Musselman
Approved by:	Paul J. Sanchez Thesis Advisor
	Susan M. Sanchez Second Reader
	James N. Eagle Chairman, Department of Operations Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Algorithms are an essential part of Operations Research (OR) methodology. Therefore, the efficiency of the algorithms must be a consideration. However, traditional approaches to assessing algorithm efficiency do not always captured the real-world trade-offs involved. This thesis explored the use of a new measure of algorithm efficiency, robustness, and contrasted it with the traditional "big-O" analysis. Sorting algorithms were used to illustrate the trade-offs.

The use of Dr. Genichi Taguchi's robust design techniques allowed us to take into account the impact of factors which would be uncontrollable in the real world, by measuring how those factors affect the consistency of the results. These factors, which are treated separately by big-O analysis, are incorporated as an integral part of robust analysis. The hypothesis was that robustness is potentially a more useful description of algorithm performance than the more traditional big-O analyses.

The results of experimentation supported this hypothesis. Where big-O analysis only considers the average performance, robustness integrates the average performance and the consistency of performance. Most importantly, the robust analysis we performed yielded results that are consistent with actual usage—practitioners prefer quicksort over heap sort, despite the fact that under big-O analysis heap sort dominates quicksort.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my wife, Martha, for her love and support throughout my time at the Naval Postgraduate School. I would like to thank my children, Ginna, Christian, and Richard for their love and the sacrifices that they have had to make for my career.

I would like to thank Professor Paul Sanchez for his guidance and patience throughout this thesis project. His assistance, enthusiasm, and professionalism benefited me from the initial proposal through the completion of this project.

I would to thank Professor Susan Sanchez. Her invaluable assistance in the development of the system used for the research and advice during the editing of this thesis greatly improved the contents of this report.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Algorithms are essential to our ability to solve problems with computing. In many cases several different algorithms yield correct results, but the amount of time required to derive those results can vary by orders of magnitude. The factors that affect the algorithm's run-time can be complex, and often are not known at implementation time—consider the situation of somebody who has to compile a software library, with no knowledge of how the library's functions may be used in the future. Analysts need a way to quantify run-time performance so they can choose between different algorithms. One measure that's widely used is the "big-O" notation. It expresses the amount of work for a given algorithm as being proportional to a bounding function, independent of implementation details such as the choice of computer hardware, operating system, programming language, compiler, etc. Despite its widespread use, big-O notation does not always yield results that are consistent with what knowledgeable programmers choose to do in practice.

In this thesis, we propose "robustness" as an alternate measure of algorithmic performance. Robustness measures overall performance in terms of a "loss function," which incorporates both average performance and consistency of performance into a single measure—smaller loss means better overall performance. With robustness, we are willing to trade off a small amount of the average performance for greater consistency across a broad range of inputs, particularly when those inputs are outside of our control.

We tested our theories on sorting algorithms—one of the best-studied areas in the field of computer science. The use of experimental design techniques allowed us to measure the impact of factors that would be uncontrollable in the real world, by measuring how those factors affect the loss.

The robust analysis yielded results that are consistent with actual usage: practitioners prefer quicksort over heap sort, despite the fact that under big-O analysis, heap sort dominates quicksort. Robust analysis captures this real-world preference. Quicksort, with a reasonable choice of pivot, yields the lowest loss of all the sorts we considered.

I. INTRODUCTION

A. BACKGROUND AND BIG "O" NOTATION

Algorithms are an essential part of Operations Research (OR) methodology. In many cases, a given problem can theoretically be solved by more than one algorithm. Therefore, OR analysts need to be aware of the efficiency of the algorithms under consideration as, in practice, the choice of algorithm can mean the difference between being able to solve a problem in a timely fashion or not. However, traditional approaches to assessing algorithm efficiency do not always capture the real-world trade-offs involved. In this thesis, we will explore the use of a new measure of algorithm efficiency, and contrast it with "big O" analysis, one of the most widely used analysis techniques. We will illustrate our points with sorting algorithms, arguably one of the best understood and most analyzed topics within the field of computer science.

Sorting algorithms are one of the most commonly used classes of algorithms employed in computers today. These algorithms arrange data by means of some well-defined ordering property, such as lexicographic ordering (for strings) or cardinality (for numerical data). The efficiency of any search is significantly enhanced through the use of sorting.

When we consider the complexity of an algorithm, we are usually concerned with properties such as the running time, or the amount of memory or intermediate storage required. This thesis will focus exclusively on running time T, but the reader should bear in mind that any other measure of interest could be used instead. T is determined by many factors, including:

- the number of elements to be processed;
- the choice of algorithm;
- the specific computing platform (combination of hardware and operating system) used;
- other tasks which may be active on the system;
- choice of programming language;

- choice of compiler; and
- the exact distribution of the input data.

Consider the position of somebody who needs to implement a software library to do sorting. When comparing sorting algorithms, the speed or run time is usually the top consideration, but other factors may be of interest, including:

- versatility in handling various data types;
- consistency of performance;
- memory requirements;
- length and complexity of code; and
- stability.

No single sorting algorithm is superior in all categories (Ring, 2003). With the exception of the choice of algorithm, most of the factors that determine the actual run time are outside of the individual's control.

In practice, it is virtually impossible to directly predict T. Instead, algorithm analysts have chosen to assume that the run time is dominated by some factor N (usually the number of elements to be processed) and tried to express T(N) as being proportional to some function of N.

The idea of proportionality is formalized by O-notation, often referred to as "big O notation," which is defined as follows. We say that a function T(N) is in O(g(N)) when there exist positive constants k and N', such that $0 <= T(N) <= k g(N)$ for all $N >= N'$. The values of k and N' must be fixed for the function T() and must not depend on N (Black, 2004). For example, $N^2 + 3N + 6$ is $O(N^2)$, since $N^2 + 3N + 6 < 2N^2$ for all $N > 10$. O-notation ignores lower order terms and constants, and expresses the resource requirements in terms of the dominant highest order term. Table 1 shows some common computing tasks, with their associated complexity and common naming conventions.

Example	Name	Notation
Determining if a number is even or odd	Constant	O(1)
Finding a particular item in a sorted list of length N using a binary search algorithm	Logarithmic	O(log N)
Finding a particular item in an unsorted list of length N by searching sequentially until it is found	Linear	O(N)
Sorting a list of length N with an efficient general-purpose algorithm such as Heap Sort or Merge Sort	N log N	O(N log N)
Sorting a list of length N with a less efficient general-purpose sort such as insertion sort; the worst-case performance of QuickSort	Quadratic	$O(N^2)$

Table 1. Examples of O-notation.

O-notation provides a proportional upper bound for the running time of an algorithm. That upper bound can be extremely loose. For example, if an algorithm has T(N) in $O(N^2)$, you can see from the definition that it is also in $O(N^3)$, $O(N^4)$, $O(N^5)$, and so on. By convention, we try to express the lowest upper bound possible when using O-notation.

A related concept is Theta notation, where $\Theta(g(N))$ denotes that kg(N) is the largest lower bound we can determine for T(N) for some value k and all N > N'. When the upper and lower bounds are the same, we can describe T(N) using Omega notation, $\Omega()$. We say that T(N) is in $\Omega(g(N))$ when T(N) is in both O(g(N)) and $\Theta(g(N))$. Rigorously determining the bounding functions is challenging for many algorithms, so Omega notation is used much less frequently than O-notation.

Algorithm analysis is cognizant of the fact that T(N) may still depend heavily on factors such as the initial distribution of the input data. The impact of such factors is not captured directly by big-O analysis—instead, what is done in practice is to analyze best-, worst-, and expected-performance as three separate big-O analyses. More recently, Sleator (1985) has proposed amortized analysis as an alternative to separate case analyses. Amortized analysis finds the average running time per operation over a worst-case sequence of operations. It differs from average-case performance in that probability is not involved; amortized analysis guarantees the time per operation over a worst-case scenario. It combines aspects of the worst-case and average-case analysis, which provides a measure of algorithmic efficiency that is more robust than average-case analysis and more realistic than worst-case analysis (Sleator and Tarjan, 1985). However, due to its complexity it is much less common in actual use than big-O analysis.

B. SORTING ALGORITHMS

In this paper, we will discuss and compare five specific classes of sorting algorithms: Insertion Sort, Shell Sort, QuickSort, Merge Sort, and Heap Sort. Sorting algorithms can be evaluated in terms of several criteria, including: run-time (best, worst, and typical); stability (whether the original order of data is preserved in the case of ties); external memory requirements; and complexity (how difficult is it to correctly implement the algorithm). Table 2 is a comparison, based on these criteria, of the five classes of sorting algorithms that we will be analyzing.

Sorting Algorithm	**Average Running Time**	**Worst Running Time**	**Stability**	**Data Types**	**Complexity**
Insertion	$O(N^2)$	$O(N^2)$	Stable	All	Very Low
Shell	$O(N (\log N)^2)$ $O(N^{1.25})$	$O(N (\log N)^2)$	Not Stable	All	Low
Quick	$O(N \log N)$	$O(N^2)$	Not Stable	All	High
Heap	$O(N \log N)$	$O(N \log N)$	Not Stable	All	Medium
Merge	$O(N \log N)$	$O(N \log N)$	Stable	All	Medium

Table 2. Performance characteristics of several sorting algorithms.

Insertion sort considers the elements one at a time, inserting each element in its proper place among those elements already positioned. It is simple, flexible, and stable, and it is fast for small sets of data. It has average- and worst-case performances of $O(N^2)$. The number of comparisons and exchanges used are about $(N^2)/4$ and $(N^2)/8$, respectively, on average, and twice as many in the worst case. However, its best-case performance is O(N)—insertion sort is linear for sorted or "almost sorted" data (Sedgewick, 1990, pp. 103-104).

Shell sort is a simple extension of insertion sort, which gains in efficiency by allowing exchanges of elements that are initially far apart. It does this by insertion-sorting every i^{th} element of the set using insertion sort, then repeating the process using smaller i values until $i = 1$. Shell sort never does more than $N^{3/2}$ comparisons (Sedgewick, 1990, p. 111) and with careful selection for the sequence of i's can do substantially better. It is also notoriously difficult to find tight bounds for shell sort, and the existence of a sequence of i's yielding O(N log N) performance is an unresolved research question.

Quicksort is the name given to a family of algorithms that partitions the data into two subsets relative to a designated value called the *pivot*. All the elements less than or equal to the pivot value are placed in the first subset, while all elements larger than the pivot are placed in the second. This can be done "in place" by scanning from the front of the set until an element larger than the pivot is encountered, then from the rear until an element less than or equal to the pivot is found, and then exchanging the two. The process is repeated from where the two exchanged elements were found, until the two searches collide somewhere in the middle, at which time the pivot is placed at that point. The operation is repeated for each subset, until the subsets each have only one element in what is frequently called a divide-and-conquer strategy. There are several variants of quicksort, which vary in the details of how the pivot is selected at each stage and how the search and exchange is accomplished. In practice, the quicksort family is among the fastest of sorting algorithms for average performance, but poor choices for the pivot can yield unbalanced partitions and lead to very slow worst-case performance. Three common methods of picking the pivot are: 1) pick the first element in the set; 2) pick the median of the first, last, and middle elements in the set; and 3) pick the pivot randomly from the set (Sedgewick, 1990, p. 126). The second of these is fairly effective at avoiding the worst-case scenario in which each subset of size N is subdivided into subsets of size 1 and size N–1. Most implementations of quicksort are not stable. Quicksort has best- and average-case running times that are O(N log N), and uses O(1) additional storage for manipulating the data. However, when the worst-case scenario described above occurs, it has $O(N^2)$ running time. On average, quicksort is faster than heap sort or merge sort, but its worst-case run time of $O(N^2)$ makes it unattractive to a risk-averse user.

Heap sort constructs a binary heap from the initial data set. A heap can be defined recursively as a partially ordered binary tree, where the root node of the tree is greater than or equal to all values stored in its two sub-trees, which are also heaps. Thus, the root of any sub-tree within the heap is equal to the largest value in that sub-tree, and the root of the heap is equal to the largest element in the set. Heap sort then removes each root value from the heap successively, rebuilding the remaining data as a valid heap after each removal. It is not a stable sort, because the heap structure reorders its elements to establish or maintain the heap property. Its best-, worst-, and average-case performances are all O(N log N). However, in practice, its average performance is as much as twice as slow as quicksort. Like quicksort, it uses O(1) additional storage for manipulating the data. Its performance is completely insensitive to the initial distribution of the input data.

Merge sort, like quicksort, is based on a divide-and-conquer strategy. First, the data are broken down into two (nearly) equal portions, to be sorted independently of each other. These two subsets are then broken down into two further subsets, and so on, until the subsets each have only one element that, trivially, is a sorted set. The two sorted halves are then merged into a larger sorted sequence. The merging is repeated back up the recursive chain until all the (now sorted) elements are again a sorted set. Merge sort has a best-, worst-, and average-case performance that are all O(N log N). In practice, it has the second-best average run time, with quite good speed on “almost sorted” data. It is stable. The main drawback of merge sort is its need for additional storage, which makes it less suitable for “in memory” sorting. Merge sort requires O(N) additional storage for intermediate operations.

C. ROBUST ANALYSIS

Dr. Genichi Taguchi pioneered a *robust design* methodology that greatly improves quality planning and engineering productivity (Taguchi, 1987). By considering so-called *noise factors* (such as manufacturing variation and component deterioration) that are uncontrollable or controllable only at prohibitive cost, and the cost of failure in the field, robust design helps ensure customer satisfaction. Robust design focuses on improving the fundamental function of the product or process by seeking consistency of performance, in addition to reasonable expected performance. One tool for assessing robustness is the use of a *loss function*, which is used to quantify the loss incurred by the user due to deviation from target performance. There are many possible loss functions—we will use *quadratic loss*, which squares the deviations from the target performance level. Quadratic loss is tolerant of small deviations, but penalizes large deviations heavily. It is also analytically convenient, because under expectation the quadratic loss decomposes into the square of the difference between expected performance and target performance, plus the variance of the performance. Expected performance and variance are both easy to estimate, and can be analyzed separately or pooled to estimate the loss function.

Robust design is a great way to analyze complex systems because it is flexible and efficient, the solutions are realistic by design, and it facilitates continuous improvement. It is flexible since it can be adapted to study systems that are analytical, physical, or simulated. It is efficient because the "sampling plans can be chosen to keep the data requirements or the analysts' time and effort low." The solutions are kept realistic by design since the "suggested system configurations have been shown to behave well over a broad range of adverse conditions." It also facilitates continuous improvement by indicating the important causes of system loss, and guides the efforts for system optimization and improvement by conveying hidden costs to the decision makers (Sanchez, 2000).

D. RESEARCH GOALS

Someone considering which sorting algorithm to implement for a library must consider the variety of contexts in which that algorithm may be used. Average-case performance may not be the appropriate criterion to use. Many analysts are influenced by the worst-case performance when comparing algorithms. For instance, quicksort has an average-case performance that is O(N log N), but its worst-case performance is O(N^2). Alternatively, heap sort has both average- and worst-case performances that are O(N log N). If the decision were based solely on big-O analysis, heap sort should be preferable to quicksort since they have the same measure for average-case performance and heap sort’s worst-case performance dominates quicksort’s. However, in practice, people prefer quicksort, which is probably more widely used than any other (Sedgewick, 1990, p. 115). Big-O analysis must be overlooking some important aspect of performance. In this thesis, we propose robust design’s loss functions as an alternate measure of algorithm performance, one which we think confirms that people are making a rational and correct decision in picking quicksort over heap sort.

II. ROBUSTNESS

A. ROBUST DESIGN

What is robust design? It is a process intended to identify "good" systems by seeking configurations that exhibit both desirable mean performance and consistency. A robust system "must be relatively insensitive to uncontrollable sources of variation present in the system's environment." "Taguchi found that it was often more costly to control the cause of manufacturing variation than to make the process insensitive to these variations, and through the use of simple experimental designs and *loss functions*" was often able to greatly improve product performance by building in quality (Sanchez, 2000). Robust design can be thought of as a process of system optimization, where the "best" answer is not overly sensitive to small changes in the system inputs. In this thesis, the goal is to evaluate several popular sorting algorithms and identify those that are robust to uncontrollable factors such as the distribution of inputs.

Experimental design techniques can be used to observe how the expected value of the performance measure varies across several different system configurations. System configurations result from changing the settings of some or all of the parameters in the system. Statistical models based on designed experiments can provide more information about the system than can be obtained by the analysis of a few alternatives. The statistical model describes the relationship between the outcomes, usually called Y, and the input variables, denoted as Xs. A statistical model is a function that can be estimated given an experimental design and the corresponding responses, Y = f(X), where X is the vector of inputs. The importance of the statistical model is that it allows the investigation to focus on the important factors that affect the research. These factors are classified as controllable factors (sometimes called parameter design factors or simply parameters) or uncontrollable factors (also called noise factors). All factors can be controlled to a large extent in a laboratory or simulation experiment, so controllability is based on real-world system capabilities. Noise factors are those factors that cannot be controlled or can be controlled only at prohibitive cost.

Robust design incorporates the identifiable noise factors into the experimental design, e.g., by crossing a controllable factors design matrix with the uncontrollable factors design matrix. The output measures of interest from each configuration in the parameter space are then summarized across the noise factor space, to determine the parameter levels that result in the smallest loss. Quadratic loss is measured by combining the squared deviation from the target value with the estimated variance over the noise factor space for each parameter configuration.

B. ELAPSED TIME FUNCTION

System runtime is the performance characteristic that we are concerned with. Elapsed time will denote the actual "clock" runtime of the sorting algorithm. We want to know what the mean and the variance of the elapsed time is for each configuration. The Elapsed time function, T(), is represented as T(N, ρ, p, sort) where:

N	denotes the size of the array or the number of elements in it;
ρ	denotes the degree of randomness in the array (preordered to reverse ordered and completely random);
p	denotes the platform type and operating system combination (Intel compatible hardware, Windows XP/Mac OS X); and
sort	denotes the sorting algorithm to be used (Insertion, Shell, Quicksort, Heap, or Merge).

Of these four factors, one is controllable and three are uncontrollable. The sort factor is controllable since that is the choice that the real-world analyst is making. The N and ρ are continuous uncontrollable factors because the data sets to be sorted in real-world usage could come from anywhere, may or may not have some degree of preordering, and can be of any size ranging from a handful to millions of elements. The p factor is a discrete uncontrollable factor, as it is distinct to the machine and operating system where an arbitrary user may be using the sorting library.

Given that N and ρ are continuous variables, an experimental design for continuous factors can be used. The p is categorical, with two values for this thesis: MS Windows or Mac OS X operating systems running on Intel compatible hardware.

C. QUADRATIC LOSS FUNCTION

We will use a general formula of a quadratic loss function to conduct our analysis of the sorting algorithms. Let T(X) denote the elapsed time observed for a vector of factor settings X. Let τ denote the target value of Y, and L(T(X)) denote the loss function. Assuming no loss will be incurred when T achieves the ideal state, the quadratic loss function can be written as:

$$L(T(X)) = c * [T(X) - \tau]^2.$$

Then the expected loss function (where expectation is taken over the uncontrollable factors) is:

$$E(\text{loss}) = c * [\sigma^2 + (\mu - \tau)^2].$$

The quadratic loss function has been used in a variety of other settings, e.g., minimizing mean squared error (MSE) loss in the basis of linear regression. Small deviations from the target value τ will have little impact on the loss, and large deviations from τ will result in very large losses. We need to consider the mean and variability of the response, jointly.

According to Taguchi, *c* is a scaling constant used to convert losses into monetary units to facilitate comparisons of systems with different capital costs. Taguchi's "loss to society" includes the cost to the end-users of the product. The expected loss is the expected value of the monetary losses that an arbitrary user of the product is likely to suffer during the product's life span due to performance variation (Taguchi and Wu, 1980). In this model, it is not easy to define the monetary losses, so *c* is set to equal 1; it has no impact on the equation and can, therefore, be removed so analysis will focus on the relative performance of the sort algorithms. For this thesis, we will call Taguchi's "loss to society" a "loss of efficiency." The loss of efficiency would be the delays and indeterminable cost to the user if the system is unavailable or it takes an excessively long time to get the results. Such costs could be incurred, for example, by missing deadlines.

To construct an actual loss function, we must specify the target value τ. Ideally we would like a sorting algorithm to work instantaneously, i.e., $\tau = 0$. This immediately gives us a problem, though. We know that T(N) is an increasing function of N for all of our sorting algorithms, so the deviation from target portion of the loss function would yield increasing loss through no fault of the algorithm. This can be overcome, however, by using a well known theoretical property of sorting algorithms—all general purpose sorting algorithms are in $\Theta(N \log N)$. If we scale $T(N)$ by dividing it by $N \log N$, all $O(N \log N)$ sorts will be ranked by their actual relative performance, and sorts that have $O(T(N)) > N \log N$ will exhibit correspondingly poorer performance as measured by quadratic loss. The resulting quadratic loss function is:

$$\text{loss} = [(T(X) / (N \log N)) - \tau]^2.$$

Since our target value is zero, this simplifies to

$$\text{loss} = [T(X) / (N \log N)]^2.$$

III. ALGORITHM DEVELOPMENT

A. DEVELOPMENT

The five classes of sorting algorithms discussed in Chapter I were implemented in the Java programming language (Sun Microsystems, Inc., 2006). The implementations were straightforward translations of the C versions found in Sedgewick (1990). Sedgewick's original versions were specifically written to sort integers. Our implementations use Java's *Comparable* and *Comparator* interfaces in order to conform to Java common usage, and thus can sort sets of arbitrary objects. We also implemented six variants of the core algorithms for Shell sort, quicksort, merge sort, and heap sort. Source code for all 11 variants can be found in Appendix A. Each variant is described briefly below.

Shell sort's performance is strongly dependent on the choice of a sequence of "stagger distances" that determine which interleaved subsets of the data will be insertion sorted. Sedgewick's Shell sort uses stagger values $s_1 = 1$, $s_i = 3s_{i-1} + 1$ for $i > 1$ (see Appendix A.11). A variant of Shell sort (see Appendix A.10) was implemented based on Fibonacci numbers, excluding the first occurrence of 1, i.e., {1, 2, 3, 5, 8, 13,…}.

Implementing quicksort requires the programmer to choose a pivot value, which is used to partition the data into two subsets as described in Chapter I. Sedgewick's quicksort (see Appendix A.8) selects the element in the middle of the set as the pivot. Our first variant of quicksort (see Appendix A.7) uses the median-of-three method, which picks the median of the first, last, and middle elements in the set. Our second variant (see Appendix A.9), which we designated as QuickSortNaive, simply picks the first element of the set as the pivot.

Heap sort works by constructing a binary heap from the initial data set, as described in Chapter I. Sedgewick's heap sort (see Appendix A.1) starts by building a min-first heap from the rear of the data set. It then swaps the smallest heap element with the last heap location, excludes that last location from the heap, and reestablishes the heap property. This process is repeated, removing one element at each iteration, until the heap is empty. Our first variant (see Appendix A.3) starts by building a max-first heap from the front of the data set. It then swaps the largest heap element with the last heap location, excludes that last location from the heap, and reestablishes the heap property. As before, the process is repeated until the heap is empty. The second variation of heap sort (see Appendix A.2) inserts each element into an instance of Java's *PriorityQueue* class, which implements a skew-heap. The elements are then removed from the skew-heap in min-first order and placed back into the original set.

Sedgewick's implementation of merge sort (see Appendix A.6) creates two partitions whose sizes differ by no more than one, and recursively sorts them into temporary additional storage. Both partitions are stored together, with the second partition's data stored in reverse order so that the largest element in either partition acts as a sentinel value for both partitions. After they have been sorted, the two partitions are recombined via a merge operation that repeatedly removes the minimum element of the two sets and copies it back into the original set. The variant (see Appendix A.5) sorts the two partitions in place, and then merges the results into temporary storage. The resulting sorted set is then simply copied back into the original location.

B. DESIGN POINTS

The program controlling the experiments needs input for two noise factors—the size of the array to be sorted (n) and the degree of randomness of the elements within the array (ρ)—to generate a new array for each experimental configuration. The range of n is 100 to 1,000,000 elements. The input to the program was $\log_{10}(N)$, i.e., 2 for $\log_{10}(100)$ and 6 for $\log_{10}(1{,}000{,}000)$. The range of ρ is –1 to 1, where a value –1 generates a perfectly reverse ordered array and a value of 1 generates a perfectly ordered array. The ordering algorithm is based on work by Knuth (2005).

The Simulation Experiments and Efficient Designs (SEED) Center <http://harvest.nps.edu> at NPS has a mission statement that reads: "Advance the collaborative development and use of simulation and efficient designs to provide decision makers with timely insights on complex systems and operations." One of the most widely used designs for computational experiments at the SEED Center and elsewhere is the Latin Hypercube (LH) design. LH designs allow the analyst to efficiently vary many factors simultaneously. We will be using one of a special class of LH called the Orthogonal Latin Hypercube (OLH) design, created by Ang (2006). This design allows the analyst to investigate up to 512 variables in only 1,025 experiments. Precursors to these designs include those of Ye (1998), who did the original development of OLHs, and Cioppa (2002), who was able to further Ye's work by creating *Nearly* Orthogonal Latin Hypercubes (NOLHs) with improved space-filling properties (see also Cioppa and Lucas, 2007).

We used a 257 level for the two continuous noise factors $\log_{10}(N)$ and ρ, and augmented it with four extreme points: $\{(2, -1), (2, 1), (6, -1), (6, 1)\}$. These extreme points were identified as particularly important based on observations made during preliminary runs. The program had difficulty completing some of the sorting operations at these extreme points. QuickSortNaive uses the first element encountered as a pivot, which caused the program to crash at $\rho = -1$ or $\rho = 1$, since both values produce 10^6 levels of recursion. This exceeds the Java Virtual Machine's stack depth. The Insertion Sort algorithm took a prohibitive amount of time to run a single experiment with 10^6 elements except when $\rho = 1$, where insertion sort becomes O(N).

To study the impact of the platform, we used an Apple MacBook Pro laptop computer with an Intel chipset and both the Microsoft Windows XP (SP2) and Mac OS X operating systems installed. This allowed a comparison of two operating systems on identical hardware to study any impact the OS might have on algorithm performance. The computer was disconnected from the internet to minimize external influences.

C. REPLICATIONS

Variations in the elapsed time when sorting the exact same data were noted from replication to replication during early testing. Therefore, the program uses a blocked design for replications to help distinguish between within- and between-vector variations. For each design point, seven different vectors of inputs are generated. Each vector is sorted five times.

The 11 sorting algorithms are categorical. The parameter design with these 11 levels is crossed with the augmented noise factor design matrix with

261 design points, for a total of 2,871 design points. Each of these 2,871 design points was replicated 35 times, yielding 100,485 runs per platform. Since two platforms were used, the total design specified 200,970 experiments.

IV. RESULTS AND ANALYSIS

A. RESULTS

Based on Ang's finely gridded design with 257 design points and four extreme points, we initiated 35 replications for 261 design points for each of the 11 algorithms. Figures 1 and 2 display the distribution of the observations by algorithm and the actual count breakdown for each algorithm, respectively. Shortfalls are due to the issues discussed at the end of Chapter III. These caused difficulties with two of the quicksort variants and the insertion sort algorithm. We were able to complete 189,595 replications of the planned 200,970, which was sufficient to conduct a meaningful analysis. Appendix B contains three-dimensional plots showing scaled runtime versus $\log_{10}(N)$ and ρ, which confirm the known big-O characteristics of the various sorting algorithms. These results came from an orthogonal, finely gridded design that is not a space-filling design. The finely gridded design is obvious in the "X" designs that are obvious in the plots in Appendix B. Nonetheless, the design provided us with a variety of configurations that were identically tested against all the algorithms.

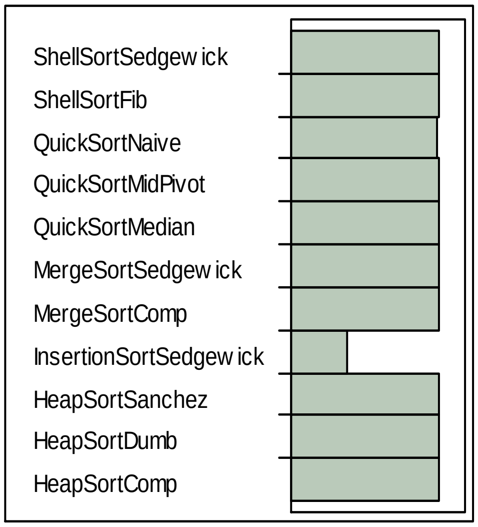

Figure 1. Observation distribution.

Level	Count
ShellSortSedgewick	18,270
ShellSortFib	18,270
QuickSortNaive	18,130
QuickSortMidPivot	18,270
QuickSortMedian	18,200
MergeSortSedgewick	18,270
MergeSortComp	18,270
InsertionSortSedgewick	7,105
HeapSortSanchez	18,270
HeapSortDumb	18,270
HeapSortComp	18,270
Total	189,595

Figure 2. Observation counts.

We began by using the design points on the Microsoft Windows platform. The InsertionSortSedgewick algorithm did not complete 58 configuration design points, due to the algorithm requiring prohibitively large run times as $\log_{10}(N)$ increased to greater than five. The data from the 7,105 replications that had been completed for this algorithm represented adequate proof that the insertion sort algorithm was indeed $O(N^2)$. This supported the well-known and well-documented fact that insertion sort is extremely slow when the number of elements to be sorted becomes very large. So, we decided to cut these design points out of the 261 configuration design points on the Microsoft Windows platform for this algorithm, and decided not to include the InsertionSortSedgewick algorithm during the Mac OS X platform runs.

Prior to discontinuing the effort of testing the InsertionSortSedgewick algorithm, we were able to complete several design points with $\log_{10}(N)$ greater than five. Each of these replications took as much as 178 seconds to complete. The algorithm with the next worst average performance, the ShellSortFib, took less than 3.5 seconds to complete each replication. The three-dimensional plots provided in Appendix B show that when n is small, say 100 or less, insertion sort performs very well. However, as n gets bigger, the time it takes to sort a set of elements becomes very large.

Additionally, we observed that 18 of the 189,595 observations were extreme outliers, between 6 and 20 times larger than any other value for their respective algorithms. A comparison was done to the other replicates with same design point (N, ρ, sort) as the outliers. The other replicates were relatively consistent among themselves. We had anticipated this might occur, given the flaws associated with the methodology of measuring elapsed time, and set the number of replications intentionally high to accommodate this possibility. This allowed us to exclude the 18 extreme outliers listed in Table 3 from the analysis without compromising the results.

Observations Excluded From Analysis			
algorithm	**log_n**	**p_rho**	**scaled time**
HeapSortComp	2.875	-0.5546	2.41E-06
HeapSortComp	2.0625	-0.9608	1.37E-05
HeapSortDumb	2.03125	-0.9765	2.87E-05
HeapSortDumb	2.04688	0.98428	7.99E-06
HeapSortSanchez	2.65625	-0.664	2.41E-06
HeapSortSanchez	2.67188	0.67181	3.60E-06
HeapSortSanchez	2	1	6.72E-06
HeapSortSanchez	2	1	6.59E-06
MergeSortComp	3.8125	-0.0859	1.83E-06
MergeSortComp	3.79688	0.10936	1.99E-06
MergeSortComp	2.57813	0.71868	3.02E-06
MergeSortComp	2.53125	-0.7265	8.80E-06
MergeSortComp	2.54688	0.7343	2.16E-06
MergeSortSedgewick	2.45313	0.78117	8.52E-06
QuickSortMedian	2.21875	-0.8827	3.82E-06
ShellSortFib	2.42188	0.7968	8.40E-06
ShellSortFib	2.39063	0.81242	8.94E-06
ShellSortFib	2.35938	0.82804	4.89E-06

Table 3. This table lists the 18 extreme outlier values that were inconsistent with the outcomes of other replicate of the same design points T(N, ρ , sort).

B. ANALYSIS

Recall that we decided to use quadratic loss, which squares the deviations from the target performance level for our analysis. To estimate the loss, we first converted the elapsed time T(N, ρ, p, sort) to scaled time for each observation:

$$\text{scaled time} = T(N, \rho, p, \text{sort}) / (N \log N).$$

This was then converted to loss:

$$\text{loss} = [\text{scaled time}]^2.$$

Since E[loss] = [variance + mean^2] when the target value is zero (as in our case), we were able to create a simple estimator of expected loss for each of our sorting algorithms by calculating both the sample variance and the squared sample mean, and summing them. Averaging was performed across both the noise factors and the replications.

Table 4 summarizes the resultant sample mean, sample variance, and estimated expected loss for each algorithm. Our analysis focuses on the loss, which incorporates the average and the consistency of performance. Therefore, Table 4 is arranged to show greatest to smallest loss. The insertion sort's loss was much greater than that of any other algorithm by four orders of magnitude. In fact, the results for InsertionSortSedgewick are underestimated since they omitted the very worst performances. This was due to the exclusion of the algorithm from the runs with the largest N values. For that reason, the insertion sort will be omitted from further comparative analysis.

Algorithm	N Rows	Mean	Variance	Loss	Relative Loss
InsertionSortSedgewick	7,105	2.51E-05	1.78E-09	2.41E-09	105,240.17
ShellSortFib	18,267	3.08E-07	3.48E-14	1.30E-13	5.68
HeapSortDumb	18,268	2.38E-07	5.94E-14	1.16E-13	5.07
QuickSortNaive	18,130	1.45E-07	4.44E-14	6.53E-14	2.85
HeapSortSanchez	18,266	2.07E-07	1.50E-14	5.79E-14	2.53
MergeSortSedgewick	18,269	1.67E-07	2.25E-14	5.05E-14	2.21
ShellSortSedgewick	18,270	1.93E-07	9.64E-15	4.68E-14	2.04
QuickSortMidPivot	18,270	1.29E-07	2.98E-14	4.63E-14	2.02
HeapSortComp	18,268	1.80E-07	9.72E-15	4.22E-14	1.84
MergeSortComp	18,265	1.58E-07	9.99E-15	3.49E-14	1.52
QuickSortMedian	18,199	1.32E-07	5.53E-15	2.29E-14	1.00

Table 4. Scaled time loss from largest to smallest, by sorting algorithm.

From Table 4, it can be seen that the variant of Sedgewick's Shell sort implemented based on Fibonacci numbers had a much greater loss than the Sedgewick's original sequence of stagger distances. While there are other methods of creating the sequence of stagger distances, Sedgewick's sequence yields an expected loss performance which is better than that of the naive quicksort, two of the heap sorts, and one of the merge sorts. ShellsortSedgewick's loss was competitive with quicksort using the midpoint as pivot. The Relative Loss column in Table 4 shows how the other algorithms compared to the QuickSortMedian.

Sedgewick's heap sort, HeapSortComp, which started by building a min-first heap from the rear of the data set, incurred the smallest loss out of the three heap sort algorithms, and it was third out of all the algorithms developed in the system. Our first variant, HeapSortSanchez, which started by building a max-first heap from the front of the data set, had the second smallest loss out of the heap sorts. The second variant of heap sort, HeapSortDumb, was the worst of the heap sorts and suffered the third largest loss of any of the algorithms. Sedgewick's heap sort not only had the lowest loss of the three, it also had the best average time and consistency; it should be implemented as the first choice if a heap sort is used.

Sedgewick's merge sort, MergeSortSedgewick, which created two partitions and sorted them into temporary additional storage using recursion on each partition, had a loss that was 30 percent greater than that of its variant. The variant, MergeSortComp, which sorted the two partitions in place and then merged the results into temporary storage, had the second lowest loss of all the algorithms. While MergeSortComp's average was only slightly lower than that of MergeSortSedgewick, it was much more consistent in its performance, resulting in a smaller loss.

Sedgewick's quicksort, QuickSortMidPivot, that selected the element in the middle of the set as the pivot, incurred a loss that was nearly double that of our first variant, QuickSortMedian. QuickSortMedian uses the median-of-three method, which picks the median of the first, last, and middle elements in the set. It had the smallest loss of all the sorting algorithms, including the best heap or merge sorts. Our second variant, QuickSortNaive, simply picks the first element of the set as the pivot. It had one of the largest resultant losses, but still had a lower loss than that of HeapSortDumb.

As discussed in Chapter I, big-O analysis suggests that an analyst should use the heap sort algorithm in preference to quicksort. Heap sort's best-, average- and worst-case performance are all O(N log N)—quicksort does no better for best- and average-case, and does worse for worst-case. Looking at Table 4, it is obvious that when measured by expected loss, all three of the quicksort algorithms perform better than any of the heap sort algorithms. The median-of-three version of quicksort performed better than any of the other algorithms we tested, when measured by expected loss. It is well known that quicksort is faster than heap sort in the average case (Sedgewick, 1990). While the median-of-three quicksort, QuickSortMedian, did not have the "best" average scaled time, its performance was so consistent over a broad range of situations that its smaller variance yielded the smallest loss of all the algorithms in our experiment. This is a significant demonstration of the concept of robustness—the analyst includes the consistency of performance as a measure of overall performance. Where big-O analysis only considers the average performance, robustness pools the average performance and the consistency of performance in the loss function.

An unexpected result was how well the ShellSortSedgewick algorithm performed. It ranks as having the fifth smallest loss, it was competitive with QuickSortMidPivot and MergeSortSedgewick, and its loss was smaller than two of the three heap sorts. This is surprising since big-O analysis states that any Shell sort should be significantly worse than a heap, merge, or quick sort. The ShellSortSedgewick sample had the fifth worst average performance, but was second in its consistency of performance. This demonstrates that the Shell sort is, in practice, quite a reliable sorting algorithm. From the plots in Appendix B, it can be seen that the Shell sort algorithms perform extremely well for smaller values of N, even when reverse ordered.

These results are summarized using boxplots in Figure 3. Recall that boxplots show the minimum, lower quartile, median, upper quartile, and maximum in a concise graphical form (Devore, 2004). Figure 3 shows parallel boxplots of the ten algorithms we studied. (Insertion sort was excluded to avoid distorting the scale of comparison, since its performance is orders of magnitude worse than all of the other algorithms.) Parallel boxplots allow us to do side-by-side comparisons of the different algorithms. Included on the boxplots are the mean diamonds. The mean diamond shows the sample mean of the data as a vertical bar within the diamond, while the points of the diamond indicate the range of a 95 percent confidence interval. Due to the large number of observations for each algorithm, the mean diamonds tend to be very tight. The dashed oval on the boxplots encompasses the three mean diamonds for the heap sort algorithms. The dotted oval encompasses the three mean diamonds for the quicksort algorithms.

Table 5 shows the numerical values of the scaled time median, and the upper and lower quartiles that correspond to the boxplots in Figure 3. It also shows the minimum and maximum sample values for each algorithm. Table 5 was provided for numerical support to the visual statistical view of Figure 3.

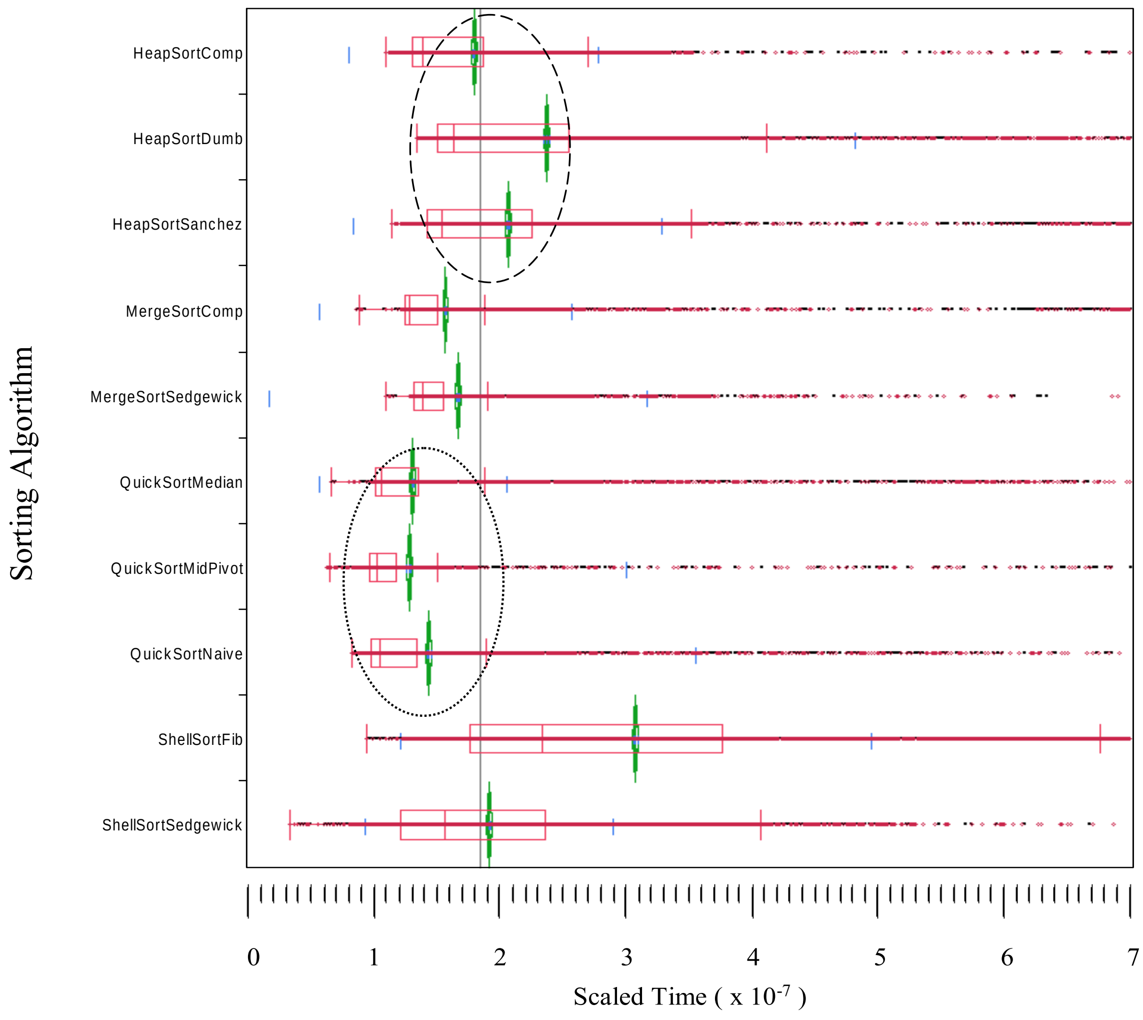

Figure 3. Parallel boxplots for comparison of sorting algorithm scaled time.

Quartiles					
Level	**Minimum**	**25% (Lower)**	**Median**	**75% (Upper)**	**Maximum**
HeapSortComp	1.10E-07	1.32E-07	1.40E-07	1.88E-07	1.57E-06
HeapSortDumb	1.35E-07	1.51E-07	1.64E-07	2.56E-07	5.25E-06
HeapSortSanchez	1.15E-07	1.43E-07	1.55E-07	2.27E-07	1.59E-06
MergeSortComp	8.69E-08	1.26E-07	1.30E-07	1.51E-07	2.33E-06
MergeSortSedgewick	1.11E-07	1.33E-07	1.40E-07	1.56E-07	6.46E-06
QuickSortMedian	6.69E-08	1.02E-07	1.07E-07	1.37E-07	1.49E-06
QuickSortMidPivot	6.41E-08	9.78E-08	1.03E-07	1.19E-07	3.87E-06
QuickSortNaive	8.40E-08	9.88E-08	1.06E-07	1.35E-07	3.44E-06
ShellSortFib	9.50E-08	1.77E-07	2.34E-07	3.77E-07	1.47E-06
ShellSortSedgewick	3.50E-08	1.22E-07	1.57E-07	2.37E-07	8.78E-07

Table 5. Quartiles.

Figure 3 supports our earlier discussion. The parallel boxplots give an indication of the distribution and variability of the outcomes for each algorithm. It is clear from the length of the boxplots that the outcomes of the three quicksort algorithms are much more consistent, or densely packed, with much smaller variability than the heap sort algorithms. All of the algorithms are skewed to the right, indicating that their scaled run times are concentrated to the left side of the plot. However, the long tails show how outcomes can be spread out, indicating variability in the outcomes. This occurs due to the testing of the full noise factor design matrix where n and ρ are varied to cover the spectrum of possible combinations of these factors. Typically, when n is large and ρ close to 1 or –1, the algorithms perform at their worst.

V. CONCLUSIONS

A. SUMMARY

Big-O notation was created because of a need to compare different algorithms in a platform- and implementation-independent manner. Despite its widespread use, big-O analysis has some significant shortcomings, and we have observed that actual usage sometimes differs from what big-O analysis would seem to indicate as the preferred algorithm. We have proposed robustness as an alternate method of comparing the performance of different algorithms. Our experiments used the speed of various sorting algorithms as a basis for comparison. Robustness measures performance using a loss function, which incorporates both the average performance and the consistency of that performance in the presence of uncontrollable factors.

We studied 11 different sorting algorithms using an orthogonal, finely gridded, experimental design to ensure systematic sampling over a broad range of settings for the noise factors. The use of robust design techniques allows us to take into account the impact of factors that would be uncontrollable in the real world, by measuring how those factors affect the consistency of the results. Factors that are treated as a basis for separate analysis in the big-O world view are incorporated as an integral part of robust analysis.

B. CONCLUSIONS

Our hypothesis was that robustness is potentially a more useful description of algorithm performance than the more traditional big-O analyses. The experiments we have done support this hypothesis. While big-O analysis focuses on the best-, average-, and worst-case performance of the algorithms as three separate analyses, the robust analysis integrates these in the form of a distribution of circumstances to be sampled over.

Most importantly, the robust analysis we performed yielded results that are consistent with actual usage—practitioners prefer quicksort over heap sort, despite the fact that under big-O analysis heap sort dominates quicksort. Robust analysis captures this real-world preference—quicksort, with a reasonable choice of pivot, yields the lowest loss of all the sorts we considered. In fact, this variant of quicksort, and one of the merge sorts, outperformed all three varieties of heap sort.

C. FUTURE RESEARCH

Recommendations for future research in the testing of the sorting algorithms would include investigating the impact of other programming languages, different compilers, and a broader variety of platforms. It would also be interesting to see the impact of the experimental design on the analysis. We used Ang's design, but Cioppa's NOLHs may do a better job of space-filling, and could therefore yield further insights in some circumstances.

Another potential avenue for exploration is the application of robust analysis to other important algorithms. Two possibilities are simulation event scheduling mechanisms or optimization algorithms.

APPENDIX A. JAVA PROGRAMMING LANGUAGE SOURCE CODE

A.1 HeapSortComp

```
/*
 * File: HeapSortComp
 * Created on Jul 18, 2007
 */
package oa.musselman.thesis;

import java.util.Comparator;

/**
 * @author Mark Allen Weiss
 * Revised - 07/17/07 - rdm
 */
/*
 * Heap sort constructs a binary heap from the initial data set. A heap can be
 * defined recursively as a partially ordered binary tree, where the root node
 * of the tree is greater than or equal to all values stored in its two
 * sub-trees, which are also heaps. Thus, the root of any sub-tree within the
 * heap is equal to the largest value in that sub-tree, and the root of the heap
 * is equal to the largest element in the set. Heap sort then removes each root
 * value from the heap successively, rebuilding the remaining data as a valid
 * heap after each removal. It is not a stable sort, because the heap structure
 * reorders its elements to establish or maintain the heap property. Its best-,
 * worst-, and average-case performances are all O(N log N). However, in
 * practice, its average performance is as much as twice as slow as quicksort.
 * Like quicksort, it uses O(1) additional storage for manipulating the data.
 * Its performance is completely insensitive to the initial distribution of the
 * input data.
 *
 * Sedgewick's heap starts by building a min-first heap from the rear of the
 * data set. It then swaps the smallest heap element with the last heap
 * location, excludes that last location from the heap, and reestablishes the
 * heap property. This process is repeated, removing one element on each
 * iteration, until the heap is empty.
 */
public class HeapSortComp<T> implements Sorter<T> {
	public void sort(Comparable<? super T>[] data) {
		for (int i = data.length / 2; i >= 0; i--)
			percDown(data, i, data.length);
		for (int i = data.length - 1; i > 0; i--) {
			Swapper.swap(data, 0, i);
			percDown(data, 0, i);
		}
	}
	private void percDown(Comparable<? super T>[] data, int i, int n) {
		int child;
		Comparable<? super T> tmp;
```

```
                for (tmp = data[i]; leftChild(i) < n; i = child) {
                        child = leftChild(i);
                        if (child != n - 1
                                        && data[child].compareTo((T) data[child + 1]) < 0)
                                child++;
                        if (tmp.compareTo((T) data[child]) < 0)
                                data[i] = data[child];
                        else
                                break;
                }
                data[i] = tmp;
        }
}
```

A.2 HeapSortDumb

```
/*
 * File: HeapSortDumb
 * Created on Jul 18, 2007
 */
package oa.musselman.thesis;

import java.util.Comparator;
import java.util.PriorityQueue;

/**
 * @author pjs
 */

/*
 * This variation of heap sort inserts each element into an instance of Java's PriorityQueue class,
 * which implements a skew-heap.  The elements are then removed from the skew-heap in min-
 * first order and placed back into the original set.
 */
public class HeapSortDumb<T> implements Sorter<T> {

	public void sort(Comparable<? super T>[] data) {
		PriorityQueue<Comparable<?	super	T>>	pq	=	new
PriorityQueue<Comparable<? super T>>(
				data.length);
		for (Comparable<? super T> element : data)
			pq.add(element);
		for (int i = 0; !pq.isEmpty(); ++i)
			data[i] = pq.poll();
	}
}
```

A.3 HeapSortSanchez

```
/*
 * File: HeapSortSanchez
 * Created on Jul 18, 2007
 */
package oa.musselman.thesis;

import java.util.Comparator;

/**
 * @author pjs
 * Revised - 07/17/07 - rdm
 */
/*
 * This sort isn't the fastest, but it's a very reliable performer. It is
 * an O(nlogn) sort for both average and worst case. The bad news is that
 * it's also O(nlogn) for already sorted data! It works by building a Heap
 * data structure, which is a partially ordered binary tree which can be
 * easily maintained in an array. The heap has the property that at each
 * vertex, the value stored is at least as large as all values in the
 * subtree for which the vertex is root. Thus, the largest element the
 * data set is in location 0, which is the root of the whole tree. This
 * maximum value gets swapped with the last location, the data set is made
 * smaller by one (which excludes the max from consideration), and the
 * remaining data are re-heaped. This brings the max of the remainder to
 * location 0. The process is repeated until there's only one element in
 * the heap. At this point, all the values which have been swapped out of
 * the heap are sorted.
 *
 * This is a completely iterative version.
 *
 * Author: Paul Sanchez, Indalo Software You may freely copy, distribute
 * and reuse the code in this example. The author disclaims any warranty
 * of any kind, expressed or implied, as to its fitness for any particular
 * use.
 */

public class HeapSortSanchez<T> implements Sorter<T> {

  @SuppressWarnings("unchecked")
  public void sort(Comparable<? super T>[] data) {

    int currentSize;
    int numElements = data.length;
    int i, current;
    int bigChild;
    int child1, child2;

    currentSize = numElements;
    /* First we convert the array to a heap */
    for (i = (numElements >> 1) - 1; i >= 0; --i) {
      for (current = i;; current = bigChild) {
        child1 = (current << 1) + 1;
        child2 = child1 + 1;
        if (child1 >= numElements)
          break;
        else if (child2 >= numElements)
          bigChild = child1;
        else if (data[child1].compareTo((T) data[child2]) < 0)
          bigChild = child2;
        else
          bigChild = child1;
        if (data[bigChild].compareTo((T) data[current]) < 0)
          break;
        Swapper.swap(data, current, bigChild);
      }
    }
    /* Then we repeatedly swap the max out and reheap the rest */
    while (currentSize-- > 1) {
      Swapper.swap(data, currentSize, 0);
      for (current = 0;; current = bigChild) {
        child1 = (current << 1) + 1;
        child2 = child1 + 1;
        if (child1 >= currentSize)
          break;
        else if (child2 >= currentSize)
          bigChild = child1;
        else if (data[child1].compareTo((T) data[child2]) < 0)
          bigChild = child2;
        else
          bigChild = child1;
        if (data[bigChild].compareTo((T) data[current]) < 0)
          break;
        Swapper.swap(data, current, bigChild);
      }
    }
  }
}
```

A.4 InsertionSortSedgewick

```
/*
 * File: InsertionSortSedgewick
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;

/**
 * @author Sedgewick
 * Revised - 07/17/07 - rdm
 */
/*
 * Insertion sort considers the elements one at a time, inserting each element
 * in its proper place among those elements already positioned.
 */
import java.util.Comparator;

public class InsertionSortSedgewick<T> implements Sorter<T> {
	public void sort(Comparable<? super T>[] data) {
		is(data);
	}

	@SuppressWarnings("unchecked")
	private void is(Comparable<? super T>[] data) {
		for (int i = 1; i < data.length; i++) {
			Comparable<? super T> tmp = data[i];
			for (int j = i; j > 0 && tmp.compareTo((T) data[j - 1]) < 0; j--) {
				data[j] = data[j - 1];
				data[j - 1] = tmp;
			}
		}
	}
}
```

A.5 MergeSortComp

```
/*
 * File: MergeSortComp
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;

import java.util.Comparator;

/**
 * @author Mark Allen Weiss
 * Revised - 07/18/07 - rdm
 */
/*
 * Merge sort, like quicksort, is based on a divide-and-conquer strategy. First,
 * the data are broken down into two (nearly) equal portions, to be sorted
 * independently of each other. These two subsets are then broken down into two
 * further subsets, and so on, until the subsets each have only one element
 * that, trivially, is a sorted set. The two sorted halves are then merged into
 * a larger sorted sequence. The merging is repeated back up the recursive chain
 * until all the (now sorted) elements are again a sorted set. Merge sort has a
 * best-, worst-, and average-case performance that are all O(N log N). In
 * practice, it has the second-best average run time, with quite good speed on
 * “almost sorted” data. It is stable. The main drawback of merge sort is its
 * need for additional storage, which makes it less suitable for “in memory”
 * sorting, is that merge sort requires O(N) additional storage for intermediate
 * operations.
 *
 * This variant sorts the two partitions ”in-place,” then merges the results
 * into temporary storage. The resulting sorted set is then simply copied back
 * into the original location.
 */
public class MergeSortComp<T> implements Sorter<T> {
	@SuppressWarnings("unchecked")
	private static Comparable[] tmpArray;

	public void sort(Comparable<? super T>[] data) {
		tmpArray = new Comparable[data.length];
		ms(data, 0, data.length - 1);
	}

	private void ms(Comparable<? super T>[] data, int left, int right) {

		if (left < right) {
			int middle = (left + right) / 2;

			ms(data, left, middle);
			ms(data, middle + 1, right);
			merge(data, left, middle + 1, right);
		}
	}

	@SuppressWarnings("unchecked")
	private void merge(Comparable<? super T>[] data, int leftPos, int rightPos,
                int rightEnd) {
        int leftEnd = rightPos - 1;
        int tmpPos = leftPos;
        int numElements = rightEnd - leftPos + 1;

        // Main loop
        while (leftPos <= leftEnd && rightPos <= rightEnd)
                if (data[leftPos].compareTo((T) data[rightPos]) <= 0)
                        tmpArray[tmpPos++] = data[leftPos++];
                else
                        tmpArray[tmpPos++] = data[rightPos++];

        while (leftPos <= leftEnd)
                // Copy rest of first half
                tmpArray[tmpPos++] = data[leftPos++];

        while (rightPos <= rightEnd)
                // Copy rest of right half
                tmpArray[tmpPos++] = data[rightPos++];

        // Copy tmpArray back
        for (int i = 0; i < numElements; i++, rightEnd--)
                data[rightEnd] = tmpArray[rightEnd];

    }
}
```

A.6 MergeSortSedgewick

```
/*
 * File: MergeSortSedgewick
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;

import java.util.Comparator;

/**
 * @author Sedgewick
 * Revised - 07/20/07 - rdm
 */
/*
 * Merge sort, like quicksort, is based on a divide-and-conquer strategy. First,
 * the data are broken down into two (nearly) equal portions, to be sorted
 * independently of each other. These two subsets are then broken down into two
 * further subsets, and so on, until the subsets each have only one element
 * that, trivially, is a sorted set. The two sorted halves are then merged into
 * a larger sorted sequence. The merging is repeated back up the recursive chain
 * until all the (now sorted) elements are again a sorted set. Merge sort has a
 * best-, worst-, and average-case performance that are all O(N log N). In
 * practice, it has the second-best average run time, with quite good speed on
 * “almost sorted” data. It is stable. The main drawback of merge sort is its
 * need for additional storage, which makes it less suitable for “in memory”
 * sorting, is that merge sort requires O(N) additional storage for intermediate
 * operations.
 *
 * Sedgewick’s implementation of merge creates two partitions whose sizes differ
 * by no more than one, and sorts them into temporary additional storage using
 * recursion on each partition. The second partition’s data are stored in reverse
 * order, so that the largest element in either partition acts as a sentinel
 * value for both partitions. After they’ve been sorted, the two partitions are
 * recombined via a merge operation that repeatedly removes the minimum element
 * of the two sets and copies it back into the original set.
 */
public class MergeSortSedgewick<T> implements Sorter<T> {
	@SuppressWarnings("unchecked")
	private static Comparable[] b;

	public void sort(Comparable<? super T>[] data) {

		b = new Comparable[data.length];
		ms(data, 0, data.length - 1);
	}

	private void ms(Comparable<? super T>[] data, int left, int right) {

		if (right > left) {
			int middle = (left + right) / 2;

			ms(data, left, middle);
			ms(data, middle + 1, right);
			merge(data, left, middle, right);
```

```
        }
    }

    @SuppressWarnings("unchecked")
    private void merge(Comparable<? super T>[] data, int left, int middle,
            int right) {
        // base case: second sublist is empty
        if (middle + 1 > right)
            return;

        // copy a[l..m] to b[l..m]
        for (int i = left; i < middle + 1; i++) {
            b[i] = data[i];
        }
        // copy a[m+1..r] to b[m+1..r] in reverse order
        for (int i = middle + 1; i < right + 1; i++) {
            b[i] = data[right + middle + 1 - i];
        }
        // merge b[l..m] with b[m+1..r] to a[l..r]
        int k = left;
        int j = right; // pointer wandering from outside inward
        for (int i = left; i < right + 1; i++) {
            if (b[k].compareTo(b[j]) > 0) {
                data[i] = b[j--];
            } else {
                data[i] = b[k++];
            }
        }
    }
}
```

A. 7 QuickSortMedian

```
/*
 * File: QuickSortMedian
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;

import java.util.Comparator;

/**
 * @author pjs, modified by rdm
 */
/*
 * QuickSort is the name given to a family of algorithms that
 * partitions the data into two subsets relative to a designated value
 * called the pivot. All the elements less than or equal to the
 * pivot value are placed in the first subset, while all elements
 * larger than the pivot are placed in the second. This can be done
 * “in-place” by scanning from the front of the set until an
 * element larger than the pivot is encountered, then from the rear
 * until an element less than or equal to the pivot is found, and then
 * exchanging the two. The process is repeated from where the two
 * exchanged elements were found, until the two searches collide
 * somewhere in the middle, at which time the pivot is placed at that
 * point. The operation is repeated for each subset, until the
 * subsets each have only one element in what is frequently called a
 * divide-and-conquer strategy.
 *
 * This variant of quicksort uses the median-of-three method, which
 * picks the median of the first, last and middle elements in the set.
 */

public class QuickSortMedian<T> implements Sorter<T> {

	@Override
	public void sort(Comparable<? super T>[] data) {
		qs(data, 0, data.length - 1);
	}

	/*
	 * This function is similar to the qsort function on page 110 of K&R.
	* it performs the quicksort algorithm on the contents of array v.
	*/
	private void qs(Comparable<? super T>[] data, int left, int right) {
		if (left >= right)
			return;
		Swapper.swap(data, left, median(data, left, (left + right) / 2, right));
			// create
		int last = left;
		for (int i = left + 1; i <= right; i++)
			if (data[i].compareTo((T) data[left]) < 0)
				Swapper.swap(data, ++last, i);
		Swapper.swap(data, left, last);
		qs(data, left, last - 1);
```

```
                qs(data, last + 1, right);
        }

        private int median(Comparable<? super T>[] data, int i, int j, int k) {
                if (((data[i].compareTo((T) data[j]) < 0) && (data[j]
                                .compareTo((T) data[k]) < 0))
                                || ((data[k].compareTo((T) data[j]) < 0) && (data[j]
                                                .compareTo((T) data[i]) < 0))) {
                        return j;
                } else if ((data[i].compareTo((T) data[k]) < 0 && data[k]
                                .compareTo((T) data[j]) < 0)
                                || (data[j].compareTo((T) data[k]) < 0 && data[k]
                                                .compareTo((T) data[i]) < 0)) {
                        return k;
                } else {
                        return i;
                }
        }
}
```

A. 8 QuickSortMidPivot

```java
/*
 * File: QuickSortMidPivot
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;
import java.util.Comparator;

/**
 * @author pjs
 */

/*
 * QuickSort is the name given to a family of algorithms that partitions the
 * data into two subsets relative to a designated value called the pivot. All
 * the elements less than or equal to the pivot value are placed in the first
 * subset, while all elements larger than the pivot are placed in the second.
 * This can be done “in-place” by scanning from the front of the set until an
 * element larger than the pivot is encountered, then from the rear until an
 * element less than or equal to the pivot is found, and then exchanging the
 * two. The process is repeated from where the two exchanged elements were
 * found, until the two searches collide somewhere in the middle, at which time
 * the pivot is placed at that point. The operation is repeated for each subset,
 * until the subsets each have only one element in what is frequently called a
 * divide-and-conquer strategy.
 *
 * This variant of quicksort uses the median-of-three method, which picks the
 * median of the first, last and middle elements in the set.
 */
public class QuickSortMidPivot<T> implements Sorter<T> {
	@Override
	public void sort(Comparable<? super T>[] data) {
		qs(data, 0, data.length - 1);
	}
	/* This function is similar to the qsort function on page 110 of K&R. it
	 * performs the quicksort algorithm on the contents of array v.
	 */
	@SuppressWarnings("unchecked")
	private void qs(Comparable<? super T>[] data, int left, int right) {
		if (left >= right)
			return;
		Swapper.swap(data, left, (left + right) / 2);
		int last = left;
		for (int i = left + 1; i <= right; i++)
			if (data[i].compareTo((T) data[left]) < 0)
				Swapper.swap(data, ++last, i);
		Swapper.swap(data, left, last);
		qs(data, left, last - 1);
		qs(data, last + 1, right);
	}
}
```

A. 9 QuickSortNaive

```
/*
 * File: QuickSortNaive
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;
import java.util.Comparator;

/**
 * @author pjs
 */

/*
 * QuickSort is the name given to a family of algorithms that partitions the
 * data into two subsets relative to a designated value called the pivot. All
 * the elements less than or equal to the pivot value are placed in the first
 * subset, while all elements larger than the pivot are placed in the second.
 * This can be done “in-place” by scanning from the front of the set until an
 * element larger than the pivot is encountered, then from the rear until an
 * element less than or equal to the pivot is found, and then exchanging the
 * two. The process is repeated from where the two exchanged elements were
 * found, until the two searches collide somewhere in the middle, at which time
 * the pivot is placed at that point. The operation is repeated for each subset,
 * until the subsets each have only one element in what is frequently called a
 * divide-and-conquer strategy.
 *
 * This variant, which is designated as “quicksort-naïve,” simply picks the
 * first element of the set as the pivot.
 */
public class QuickSortNaive<T> implements Sorter<T> {
	@Override
	public void sort(Comparable<? super T>[] data) {
		qs(data, 0, data.length - 1);
	}

	/*
	 * This function is similar to the qsort function on page 110 of K&R. it
	 * performs the quicksort algorithm on the contents of array v.
	 */
	private void qs(Comparable<? super T>[] data, int left, int right) {
		if (left >= right)
			return;
		int last = left;
		for (int i = left + 1; i <= right; i++)
			if (data[i].compareTo((T) data[left]) < 0)
				Swapper.swap(data, ++last, i);
		Swapper.swap(data, left, last);
		qs(data, left, last - 1);
		qs(data, last + 1, right);
	}
}
```

A. 10 ShellSortFib

```
/*
 * File: ShellSortFib
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;

import java.util.Comparator;

/**
 * @author pjs
 * Revised 07/17/07 - rdm
 */
/*
 * Shell sort's performance is strongly dependent upon the choice of a
 * sequence of "stagger distances," which determine which interleaved
 * subsets of the data will be insertion sorted. This is a variant of
 * Shell sort was implemented based on Fibonacci numbers, excluding
 * the first occurrence of 1, i.e., {1, 2, 3, 5, 8, 13,…}
 */

public class ShellSortFib<T> implements Sorter<T> {

        private static final int gap[] = { 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229,
832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169,
63245986, 102334155, 165580141, 267914296, 433494437, 701408733, 1134903170,
1836311903 };

        public void sort(Comparable<? super T>[] data) {
                int i = 0;
                int j;
                // Comparable<? super T> tmp;
                Comparable<? super T> tmp;
                while (gap[i] < data.length)
                        ++i;
                for (; i >= 0; --i) {
                        for (int current = gap[i] - 1; current < data.length;
                             ++current) {
                                tmp = data[current];
                                j = current;
                                while (j >= gap[i] && data[j –
                                                gap[i]].compareTo((T) tmp) > 0) {
                                        data[j] = data[j - gap[i]];
                                        j -= gap[i];
                                }
                                data[j] = tmp;
                        }
                }
        }
}
```

A. 11 ShellSortSedgewick

```
/*
 * File: ShellSortSedgewick
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;

import java.util.Comparator;
/**
 * @author pjs
 * Revised 07/17/07 - rdm
 */
/*
 * Shell sort’s performance is strongly dependent upon the choice of a
 * sequence of “stagger distances,” which determine which interleaved
 * subsets of the data will be insertion sorted. Sedgewick’s Shell sort
 * uses stagger values s1 = 1,si = 3si-1 + 1 for i > 1.
 */
public class ShellSortSedgewick<T> implements Sorter<T> {
  public void sort(Comparable<? super T>[] data) {
                  int gap;
                  int j;
                 Comparable<? super T> tmp;
                  for (gap = 1; gap < data.length/9; gap = 3*gap +1);
                  for ( ; gap > 0; gap /= 3){
                      for (int current = gap - 1; current < data.length;
                            ++current) {
                                tmp = data[current];
                                j = current;
                                while (j >= gap && data[j - gap].compareTo((T)
                                        tmp) > 0){
                                         data[j] = data[j - gap];
                                         j -= gap;
                                }  data[j] = tmp;
                      }
                  }
         }
}
```

A. 12 GenerateTimings

```
/*
 * File: GenerateTimings
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;

import or.utils.Timer;

/**
 * @author pjs
 */
public class GenerateTimings {

        private static final boolean DEBUG = false;

        private static int[] unsorted;

        public static void main(String[] args) {
                Sorter[] s = {

                 new HeapSortComp<Integer>(),
                 new HeapSortDumb<Integer>(),
                 new HeapSortSanchez<Integer>(),
                 new MergeSortComp<Integer>(),
                 new MergeSortSedgewick<Integer>(),
                 new ShellSortFib<Integer>(),
                 new ShellSortSedgewick<Integer>(),
                 new QuickSortMidPivot<Integer>(),
                 new QuickSortMedian<Integer>(),
                 new QuickSortNaive<Integer>(),
                 new InsertionSortSedgewick<Integer>()
                 };
                // add new method here
                Timer t = new Timer();
                double timing;
                String algorithm;

                // Parameter extraction
                int n = 1000000;
                double logn = 6.0; // Added for ease or graphing and representation
                double rho = 0.0;
                int replicationsWithin = 10;
                int replicationsBetween = 7;
                if (args.length > 0) {
                        logn = Double.parseDouble(args[0]); // Change to Double to use log_n
                        n = (int) Math.pow(10, logn); // "n" needs to be an integer
                        rho = Double.parseDouble(args[1]);
                        replicationsWithin = Integer.parseInt(args[2]);
                                // Generate same vector to be sorted

                        replicationsBetween = Integer.parseInt(args[3]);
                             // Generate new vector to be sorted
```

```
		}
		// Array generation
		// This for outer loop allows for the production of more than vector for
		// each configuration of n and rho.  Between vectors.
		System.out.println("algorithm\tlog_n\tp_rho\twithin\tbetween\ttime");
		for (int k = 1; k <= replicationsBetween; ++k) {
			unsorted = PermutationGenerator.generate(n, rho);
			Integer[] data = new Integer[unsorted.length];

			for (int i = 0; i < s.length; ++i) {
				algorithm = s[i].getClass().getSimpleName();
				// This for inner loop allows for the production of he same
				// vector for each configuration of n and rho.  Within a vector.
				for (int j = 1; j <= replicationsWithin; ++j) {
					dataCopy(data);
					t.resetTimer();
					s[i].sort(data);
					timing = t.elapsedTime();
					if (DEBUG) {
						for (int x : data)
							System.out.print(x + " ");
					    System.out.println();
					}
					System.out.println(algorithm + "\t " + logn + "\t " + rho
							+ "\t " + j + "\t " + k + "\t " + timing);
				}
			}
		}
	}

	private static void dataCopy(Integer[] d) {
		for (int j = 0; j < unsorted.length; ++j)
			d[j] = new Integer(unsorted[j]);

	}
}
```

A. 13 PermutationGenerator

```
/*
 * File: PermutationGenerator
 * Created on May 15, 2007
 */
package oa.musselman.thesis;
/**
 * @author pjs
 */
import java.util.Arrays;
import java.util.Random;
import umontreal.iro.lecuyer.probdist.NormalDist;

public class PermutationGenerator {
  private static Random r = new Random();
  private static int counter;

  private static class NormQuantiles implements Comparable<NormQuantiles> {
    private int index;
    private double z;

    public NormQuantiles(double rho, int n) {
      super();
      index = ++counter;
      z = (rho * NormalDist.inverseF01((index - 0.5) / n))
        + (Math.sqrt(1.0 - rho*rho) * r.nextGaussian());
    }

    public int compareTo(NormQuantiles arg0) {
      if (z < arg0.z) return -1;
      if (z > arg0.z) return 1;
      return 0;
    }

    public int getIndex() {
      return index;
    }
  }

  public static int[] generate(int setSize, double rho) {
    counter = 0;
    NormQuantiles[] bvn = new NormQuantiles[setSize];
    int[] result = new int[setSize];
    for (int i = 0; i < bvn.length; ++i)
      bvn[i] = new NormQuantiles(rho, setSize);
    Arrays.sort(bvn);
    for(int i = 0; i < bvn.length; ++i)
      result[i] = bvn[i].getIndex();
    return result;
  }

  /**
   * @param args
   */
  public static void main(String[] args) {
    if (args.length == 2) {
      int n = Integer.parseInt(args[0]);
      double rho = Double.parseDouble(args[1]);
      // double[] rhoValues = { 0.0, -1.0, 1.0, -0.7, 0.7, -0.9, 0.9 };
      // for (double rho : rhoValues) {
        // System.out.println("Rho = " + rho);
        for (int i : generate(n, rho))
          System.out.println(i + " ");
        // System.out.println("\n");
      // }
    } else {
      System.err.println("Must supply two args: n and rho.\n");
      System.err.println("N is the number of values to generate, rho");
      System.err.println("is the measure of randomness (-1 to 1, where");
      System.err.println("-1 is reverse non-random, 0 is completely");
      System.err.println("random, and 1 is forward order non-random).");
    }
  }
}
```

A. 14 Sorter

```
/*
 * File: Sorter
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;

import java.util.Comparator;
/**
 * @author pjs
 */
public interface Sorter<T> {
    public void sort(Comparable<? super T>[] data);
    public void sort(T[] data, Comparator<? super T> c);
}
```

A. 15 Swapper

```
/* File: Swapper
 * Created on Jul 16, 2007
 */
package oa.musselman.thesis;
/**
 * @author pjs
 */
public class Swapper {
  protected static void swap(byte[] data, int i, int j) {
    byte tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
  }

  protected static void swap(char[] data, int i, int j) {
    char tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
  }

  protected static void swap(int[] data, int i, int j) {
    int tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
  }

  protected static void swap(long[] data, int i, int j) {
    long tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
  }

  protected static void swap(float[] data, int i, int j) {
    float tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
  }

  protected static void swap(double[] data, int i, int j j) {
    double tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
  }

  protected static void swap(Object[] data, int i, int j) {
    Object tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
  }
}
```

APPENDIX B. THREE-DIMENSIONAL PLOTS

This appendix contains three-dimensional plots of the 11 algorithms, which show the scaled run time versus $\log_{10}(N)$ and ρ. Figures 4–14 demonstrate how the uncontrollable factors, $\log_{10}(N)$ and ρ, affect the run-time of the different sorting algorithms. These results came from an orthogonal, finely gridded design; this is the reason that they produced an "X" in the plots. This is not a space-filling design. Nonetheless, the design provided us with a variety of configurations that were identically tested against all the algorithms. The black dots are the outcomes generated with the Microsoft Windows operating system. The red diamonds are the outcomes generated with the Mac OS X operating system. As discussed in Chapter II, all runs were made on one Intel-compatible platform with both operating systems installed on it. Notice how the two sets of outcomes are relatively consistent. The only exception is the InsertionSortSedgewick plot (Figure 4)—it only has outcomes from the Microsoft Windows operating system.

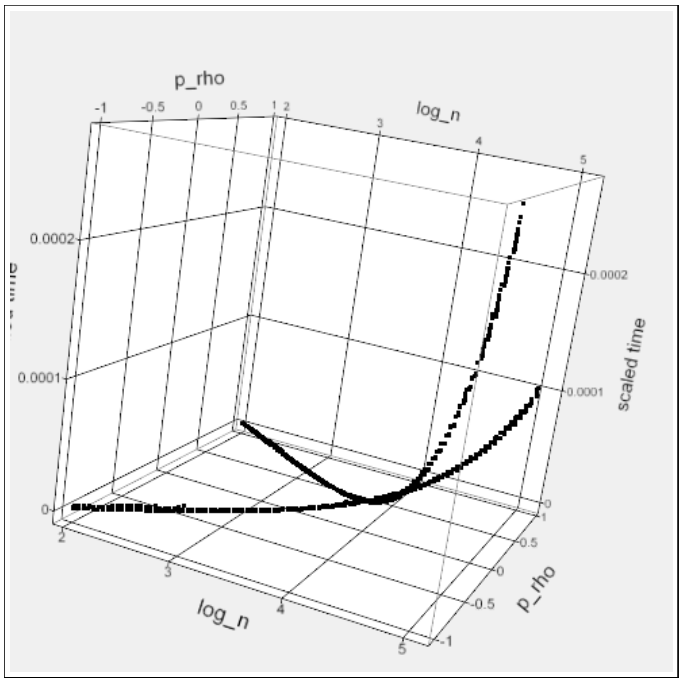

Figure 4. InsertSortSedgewick: Notice how the scaled time increases as N increases on $O(N^2)$.

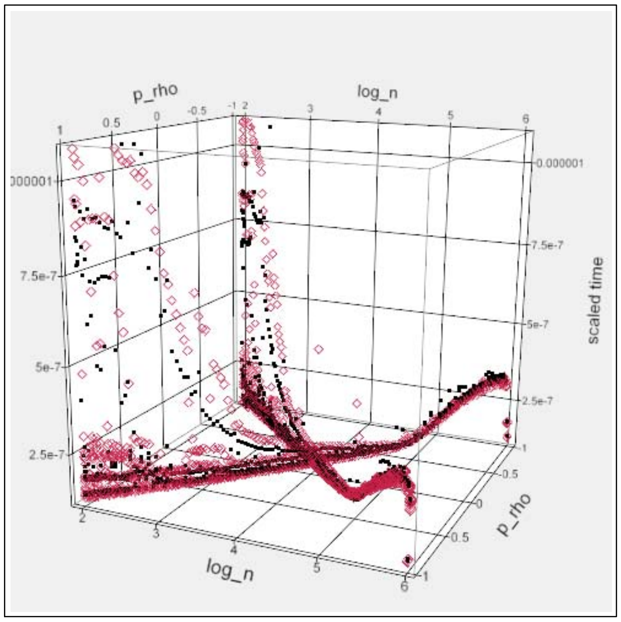

Figure 5. HeapSortComp: Notice that the largest times are where N is relatively small. Heap sort does not handle small "almost ordered" sets well.

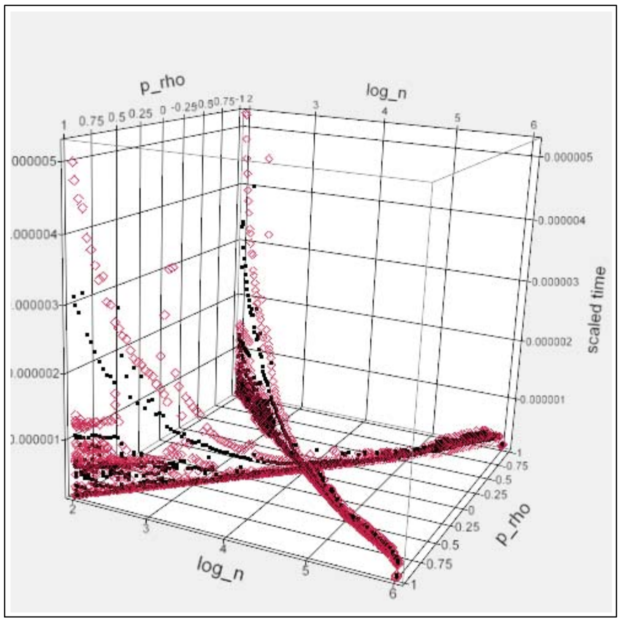

Figure 6. HeapSortDumb: Notice that the largest times are where N is relatively small. Heap sort does not handle small "almost ordered" sets well.

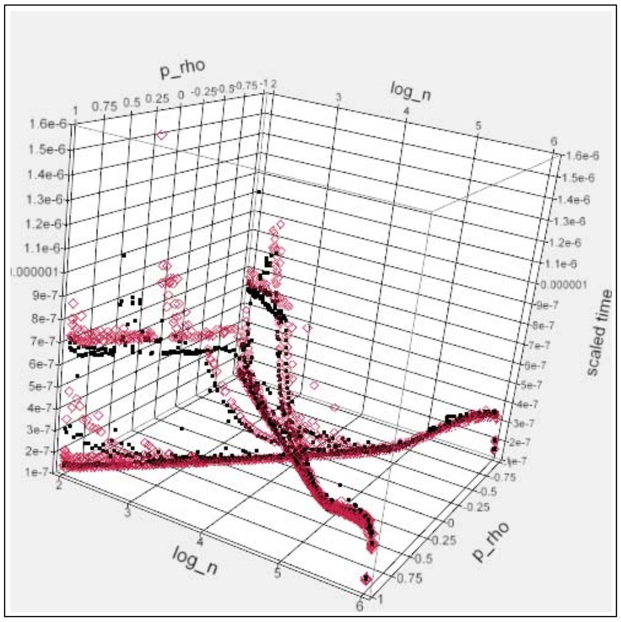

Figure 7. HeapSortSanchez: Notice that the largest times are where N is relatively small. There is significant variance.

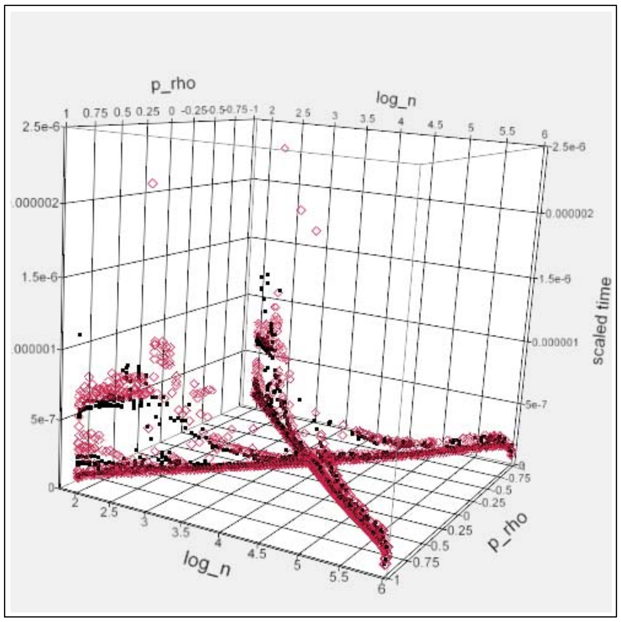

Figure 8. MergeSortComp: Does better at larger N, but suffers from a great deal of variance.

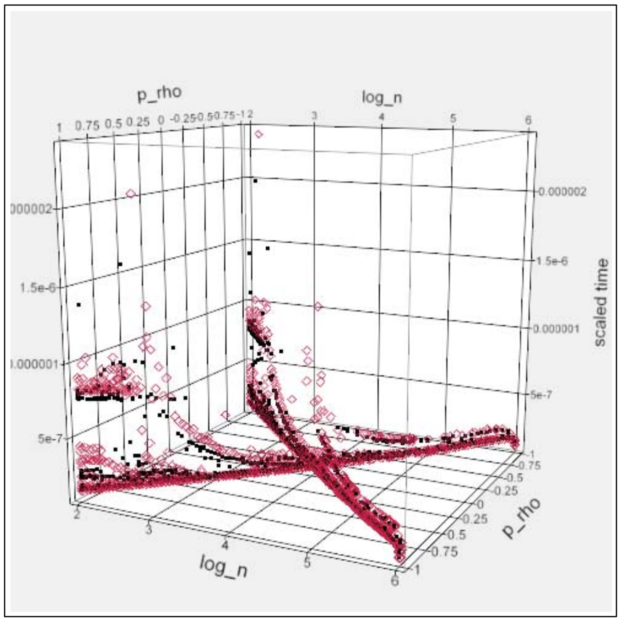

Figure 9. MergeSortSedgewick: Scaled time scale stretched to allow comparison with MergeSortComp due to numerous larger runtimes.

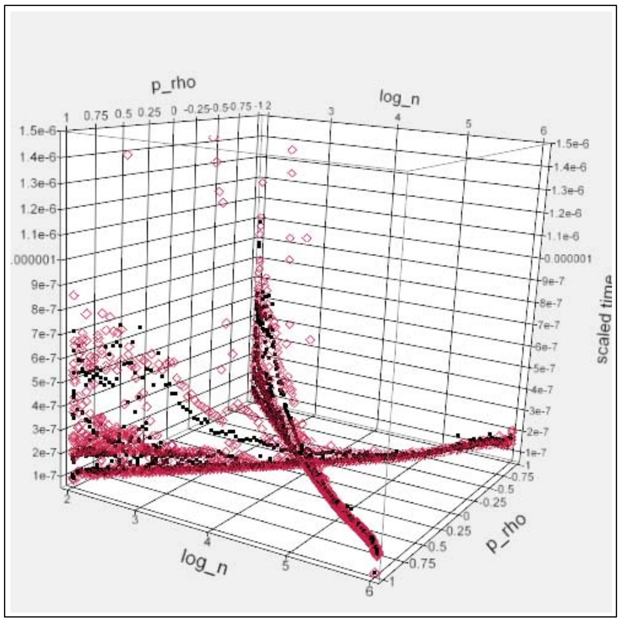

Figure 10. QuickSortMedian: Shows some degree of variability at small N, as ρ approaches –1 and 1.

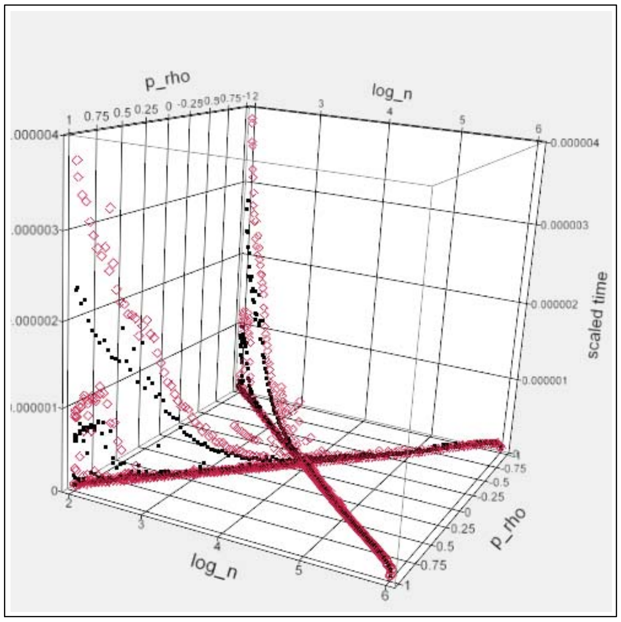

Figure 11. QuickSortMidPivot: Fairly consistent performance at large N, and actually runs faster near ρ = –1 and 1. Greater variability for small N as ρ approaches –1 and 1.

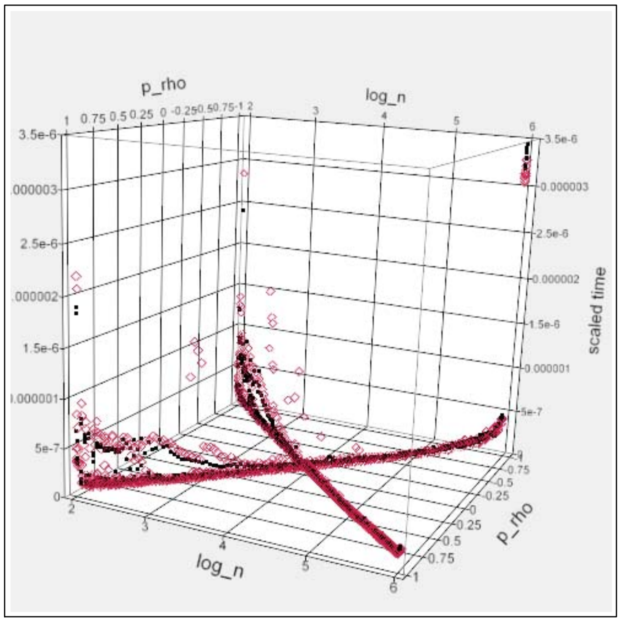

Figure 12. QuickSortNaive: The scaled time is fairly consistent, except at large N approaching ρ = –1, this is quicksort's worst-case.

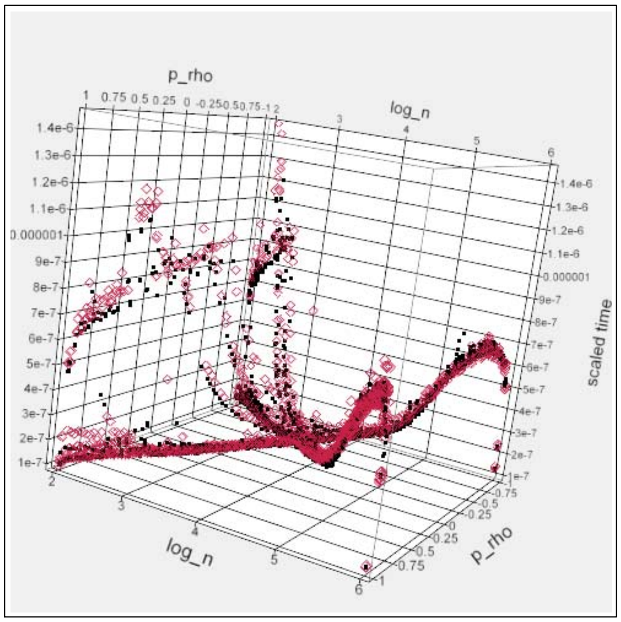

Figure 13. ShellSortFib: Shell sort does well at small N, and when randomly ordered. Notice that the scaled time increases as ρ approaches –1 and 1, but drops off sharply at –1 and 1.

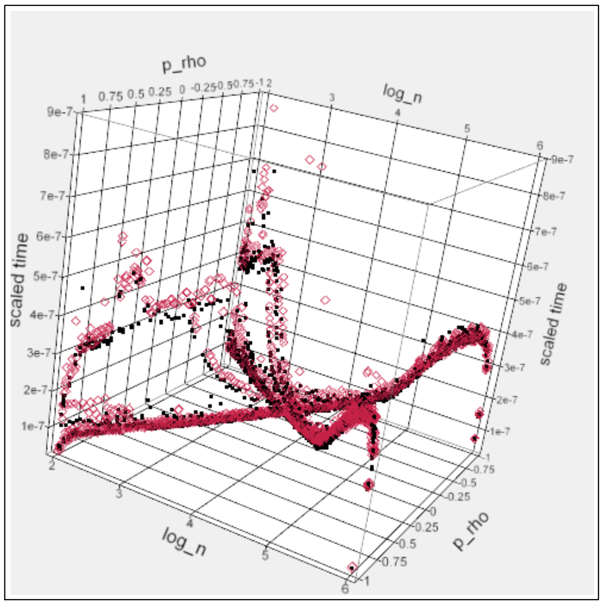

Figure 14. ShellSortSedgewick: Shell sort does well at small N, and when randomly ordered. Notice that the scaled time increases as ρ approaches –1 and 1, but drops off sharply at –1 and 1. ShellSortSedgewick is as much as twice as fast as ShellSortFib.

LIST OF REFERENCES

Ang, K-E.J., *Extending Orthogonal and Nearly Orthogonal Latin Hypercube Designs for Computer Simulation and Experimentation*, Master's Thesis, Naval Postgraduate School, Monterey, CA, 2006, pp. 14-15.

Black, P.E., *Dictionary of Algorithms and Data Structures*, United States National Institute of Standards and Technology, 2004, http://www.nist.gov/dads/, last accessed on 17 July 2007.

Cioppa, T.M., *Efficient Nearly Orthogonal and Space-Filling Experimental Designs for High-Dimensional Complex Models*, Doctoral Dissertation, Naval Postgraduate School, Monterey, CA, 2002.

Cioppa T.M.and Lucas T.W., *Efficient Nearly Orthogonal and Space-Filling Latin Hypercubes*, Technometrics, Volume 49, Number 1, February 2007, pp. 45-55.

Devore, J.L., *Probability and Statistics, For Engineering and the Sciences, Sixth Edition,* Brooks/Cole — Thomson Learning, Belmont, CA 94002, 2004, pp. 41-43.

Knuth, D., *The Art of Computer Programming*, Volume 4: *Generating All Tuples and Permutations*, Fascicle 2, first printing, Addison-Wesley Publishing Company, Inc., Reading, MA, 2005.

Ring, D.B., *A Comparison of Sorting Algorithms*, 2003, http://www.devx.com/vb2themax/Article/19900, last accessed on 7 June 2007.

Sanchez, S.M., *Robust Design: Seeking the Best of All Possible Worlds*, in Proceedings of the 2000 Winter Simulation Conference, eds. J.A. Joines, R.R. Barton, K. Kang, and P.A. Fishwick, 2000, pp. 69-76.

Schruben, L.W., Sanchez, S.M., Sanchez, P.J., and Czitrom,V.A., *Variance Reallocation in Taguchi's Robust Design Framework*, in Proceedings of the 1992 Winter Simulation Conference, eds. J.J. Swain, D. Goldsman, R.C. Crain, and J.R. Wilson, Institute of Electrical and Electronic Engineers: Piscataway, NJ, 1992, pp. 548-556.

Sedgewick, R., *Algorithms in C*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1990, pp. 100-132.

Sleator, D.D. and Tarjan, R.E., *Amortized Efficiency of List Update and Paging Rules*, Communications of the ACM, 28(2), February 1985, pp. 202-208.

Sun Microsystems, Inc., JAVA DOCs/APIs, 2006, http://java.sun.com/javase/6/docs/api/, last accessed on 31 July 2007.

Taguchi, G., *System of Experimental Design*, Vols. 1 and 2, White Plains, NY, UNIPUB/Krauss International, 1987.

Taguchi, G. and Wu, Y., *Introduction to Off-Line Quality Control*, Nagoya, Japan: Central Japan Quality Association, 1980.

Weiss, M.A., *Data Structures and Problem Solving Using JAVA*, *Third Edition,* Addison-Wesley Publishing Company, Inc., Reading, MA, 2006, pp. 313-331.

Ye, K.Q., *Orthogonal Column Latin Hypercubes and their application in computer experiments.* Journal of the American Statistical Association — Theory and Methods, 93, 1998, pp. 1430-1439.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Paul J. Sanchez
   Naval Postgraduate School
   Monterey, California

4. Susan M. Sanchez
   Naval Postgraduate School
   Monterey, California

5. James N. Eagle
   Naval Postgraduate School
   Monterey, California

6. Thomas W. Lucas
   Naval Postgraduate School
   Monterey, California